Formally Analysing a Security Protocol for Replay Attacks

Benjamin W. Long
School of Information Technology and Electrical Engineering
The University of Queensland, 4072, Australia
benl@itee.uq.edu.au

Colin J. Fidge
School of Software Engineering and Data Communications
Queensland University of Technology, 4001, Australia
c.fidge@qut.edu.au

Abstract

The Kerberos-One-Time protocol is a key distribution protocol promoted for use with Javacards to provide secure communication over the GSM mobile phone network. From inspection we suspected a replay attack was possible on the protocol. To check this, we formally specified the protocol using Object-Z and then analysed its behaviour in the presence of an attacker using the Symbolic Analysis Laboratory's model checker. To produce accurate results efficiently, our formalism included an abstraction of the protocol's data structures that captured just those characteristics that we believed made the protocol vulnerable. Ultimately, the model checker's analysis confirmed our suspicions about the protocol's weakness.

1 Introduction

The GSM (Global System for Mobile Communications) mobile phone network currently connects over two billion subscribers using wireless technology. Every mobile phone uses a Subscriber Identification Module (SIM) which is a smart card that enables the phone to access the network. Cimato [6] derived the Kerberos-One-Time protocol and suggested that its implementation could be used in conjunction with a GSM Javacard (a GSM SIM that executes Java code) to provide secure communication.

Inspection of the proposed protocol led us to believe that a *replay attack* could potentially occur. Replay attacks [29] occur when an intruder copies an encrypted message and replays it in a different protocol instance. If the receiving agent does not realise the message is from a previous instance and continues to participate in the protocol, security may be compromised.

We are in an age that requires us to follow rigorous development processes. For example, the *Common Criteria* [31], an international standard for development and evaluation of security systems, requires formal methods to be used in order to obtain the highest level of assurance (EAL7).

In this paper we take advantage of the Object-Z [11] specification language and the SAL [9] model checker to

- present a formal specification of the Kerberos-One-Time protocol;
- produce the suspected replay attack, thus verifying its existence; and
- prove that a suggested fix will prevent this or similar attacks from occurring.

2 Related work

Work already done in formal analysis of security protocols is dominated by trace-based methods such as CSP/FDR [24], the spi calculus [1], Murφ [20], and the purpose built Interrogator [19] and Brutus [7] model checkers. Logics [21, 22] including the specialised BAN logic [5] and TAPS [8] theorem prover have also been well researched. Other purpose-built methods include the NRL Protocol Analyzer [17], the AVISPA security protocol model checker [3], and strand spaces [30] supported by the Athena tool [26].

Meadows [18] states that most of the attention has been paid to the design of languages for evaluating security, whereas, the ability to specify their desired behaviour is also important. Similarly, Gollmann [12] observes that high-level abstract protocol specifications may not be precise enough for implementers.

Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)
1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

General purpose state-based formalisms such as B [2], Z [27], and VDM [4] have seldom been used for protocol analysis despite the potential for accurate modelling of message content and agent behaviour allowed by their rich, expressive data structures.

Previously, we demonstrated the value of using the Z and Object-Z formal specification languages for verifying attacks on security protocols due to the expressive nature of these notations [15, 16]. More recently, Smith and Wildman [25] showed how Z specifications can be analysed using the SAL model checker. Their research highlighted the similarities and compatibility between the formal styles of Z and SAL. Additionally, Rushby [23] has already used the SAL model checker to verify the Needham-Schroeder Protocol. In this paper we combine these principles and show how to use SAL for automated verification of an Object-Z security protocol specification.

3 Review of the Kerberos-One-Time protocol

The original Kerberos protocol [28] aimed to establish new session keys for confidential communication between protocol agents. However, subsequent communication using these keys was potentially subject to replay attacks [13]. In order to ensure replay attacks are not possible, a freshness identifier must be included in each message. The Kerberos-One-Time protocol [6] was derived from integration of the Kerberos protocol [28] and Lamport's one time password scheme [14] as a method of ensuring such freshness. The protocol consists of two phases as discussed in the following sections.

3.1 Establishing a session key

The following protocol describes the first phase in which a session key is established between two agents, Alice A and Bob B, via a trusted third party, Sam S.

1. $A \rightarrow S : A, \{B\}_{K_{AS}}$
2. $S \rightarrow A : \{B, K_{AB}, w\}_{K_{AS}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
3. $A \rightarrow B : A, \{A, H^t(w)\}_{K_{AB}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
4. $B \rightarrow A : B, \{H^t(w) + 1\}_{K_{AB}}$

Alice initiates the protocol by sending message 1 to Sam, containing her identity A and Bob's identity B, indicating that she wishes to establish a secure session key with Bob. Bob's identity is encrypted (denoted by '$\{\}$') using the key K_{AS} shared by Alice and Sam.

Sam responds to Alice's request in step 2 with two encrypted messages. The first is encrypted for Alice and contains a new randomly generated session key K_{AB} for communication with Bob and a seed w for authentication purposes. The second message (or *ticket*) encrypted for Bob contains the new session key, the first password (the value $H^t(w)$ of the seed after being hashed t times), and a timestamp T.

In step 3 Alice forwards the ticket to Bob in order to establish communication with him. She also sends an *authenticator* — a segment encrypted with the new session key K_{AB} that contains her identity A and the password. By decrypting the authenticator using the new session key and checking that the content of the authenticator is consistent with that of the ticket, Bob confirms that Alice is the initiating agent. As long as Bob stores all received authenticators for the lifetime of the ticket, the timestamp enables Bob to determine if this third message has been replayed.

Finally, Bob acknowledges the fact that he received the correct key for communication by replying with a modified version of the password in step 4. Alice can use this message to confirm that Bob has received the session key since the original password in step 3 can be accessed only by someone with key K_{BS}.

3.2 Continuation of the protocol

Subsequent communication from Alice follows in the second phase of the protocol in which Alice includes the password, this time hashed $t - i$ times where i is the ith message after initialisation.

$$A \rightarrow B : A, \{i, H^{t-i}(w)\}_{K_{AB}}$$

Bob will authenticate the message based on the one time password scheme in which he hashes the password i times and compares the result with the original password received $H^t(w)$.

When all passwords for this seed have expired, i.e., when i equals the maximum number of times the seed w may be used, Bob will not accept any more messages until the seed has been renewed. However, if a message of the following form is sent, where w' is a new seed for a new chain of one time passwords, there is no need to start the protocol from initialisation again [6].

$$A \rightarrow B : \{i, H^{t-i}(w), H^t(w')\}_{K_{AB}}$$

3.3 A potential weakness

According to Cimato, the use of a session key should be restricted to a short time period, avoiding the possibility of cryptanalytic attacks and limiting the number of messages compromised if the key is derived [6].

Bob's behaviour is consistent with this requirement as we are told that on receiving message 3, which contains the new session key, Bob checks the timestamp T to ensure the

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

message was not replayed. However, we found it strange that Alice does not make a similar check for freshness when she receives the key in step 2. This inconsistency led us to wonder whether the protocol was susceptible to a replay attack. The following sections demonstrate the approach to formal analysis taken to confirm our suspicion.

4 Specifying the protocol in Object-Z

Object-Z [11] is an object-oriented formal specification language in which set theory and logic is used to describe the internal states of system classes and the behaviour of class operations on those states. The advantage of using Object-Z for specifying security protocols is that it enables rigorous proof to be applied and it has powerful data structures that enable concise specifications to be written.

Based on the informal description of the protocol in Section 3 and our previous work on modelling protocols [15, 16], we now specify this protocol in Object-Z.

4.1 Defining data types

Protocol messages are constructed from several data items. Therefore, we assume the set of all such data items as a given type.

$$[ITEM]$$

Given the set of all items, the set of all messages MSG is the set of all possible sequences of items.

$$MSG == \text{seq}\, ITEM$$

Now we declare seven subsets of $ITEM$ for the different types of data items used in the protocol: agent identifiers AID, keys KEY, seeds SED, timestamps TSP, encrypted items ENC, hashed items HSH, and acknowledgement items ACK.

$$\begin{array}{|l} AID : \mathbb{P}\, ITEM \\ KEY : \mathbb{P}\, ITEM \\ SED : \mathbb{P}\, ITEM \\ TSP : \mathbb{P}\, ITEM \\ ENC : \mathbb{P}\, ITEM \\ HSH : \mathbb{P}\, ITEM \\ ACK : \mathbb{P}\, ITEM \\ \hline \text{disjoint}\langle AID, KEY, SED, TSP, ENC, HSH, ACK\rangle \\ AID \cup KEY \cup SED \cup TSP \cup \\ \quad ENC \cup HSH \cup ACK = ITEM \end{array}$$

Object-Z's 'disjoint' operator is used to ensure that the sets are pairwise disjoint, i.e., each element within a set is not an element of any other set. For completeness we also specify that each element in $ITEM$ must be in one of the declared sets.

4.2 Defining supporting functions

To complete the set of basic data structures defined above, we specify various functions on messages to produce encrypted, hashed, and acknowledgement items.

The strongest behaviour required of the encryption function for our model is that, given any key, a unique encrypted item will be produced for each supplied message. This behaviour is captured by the following function enc in which an injective function is used to ensure unicity.

$$enc : KEY \rightarrow (MSG \rightarrowtail ENC)$$

The one time password scheme relies on multiple hashing of the seed. To allow for this in our model we first define a hash function $hash$ that produces a unique hashed item for each given message.

$$hash : MSG \rightarrowtail HSH$$

Secondly, we define a recursive function $hash_n$ that takes as input a message and a positive integer ($n : \mathbb{N}_1$) and produces the result of hashing the message n times.

$$\begin{array}{|l} hash_n : (MSG \times \mathbb{N}_1) \rightarrowtail HSH \\ \hline \forall\, m : MSG;\ n : \mathbb{N}_1 \bullet \\ \quad n = 1 \Rightarrow hash_n(m, n) = hash(m) \land \\ \quad n > 1 \Rightarrow \\ \qquad hash_n(m, n) = hash_n(\langle hash(m)\rangle, n - 1) \end{array}$$

When $n = 1$, the function will return the result of hashing m once. Otherwise, $hash(m)$ and $n - 1$ will be used as input for another iteration of the function.

The final function ack is that which produces a related reply as an acknowledgement, corresponding to the incrementing of the password in step 4 of the protocol. Again, an injective function is used to ensure unicity.

$$ack : MSG \rightarrowtail ACK$$

4.3 Specifying protocol roles

The protocol describes three roles agents can play: Alice, Bob, and Sam. Each role is captured within a single Object-Z class specification, including only information and operations relevant to the role it is modelling. Since the replay attack is suspected to exist in the first phase of the protocol, operations belonging to the second phase are not shown here for the sake of brevity.

The first class we model corresponds to Alice's role. Alice makes use of two identities A and B, a password seed w, the key K_{AS} shared between Alice and Sam, the new session key K_{AB}, and the current message msg in transit. These variables are declared accordingly in the class state

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

schema. Initially, there is no message in transit, indicated by the empty sequence in the *INIT* schema. (We assume that upon initialisation, keys and seeds are assigned a 'random' value.)

Alice

$A, B : AID;\ w : SED$
$K_{AS}, K_{AB} : KEY$
$msg : MSG$

INIT

$msg = \langle\rangle$

requestKey

$\Delta(msg)$

$msg = \langle\rangle \land msg' = \langle A, enc(K_{AS})(\langle B\rangle)\rangle$

receiveKey

$\Delta(msg, K_{AB}, w)$

$\exists\, e : ENC \bullet$
$msg = \langle enc(K_{AS})(\langle B, K_{AB}{}', w'\rangle), e\rangle \land$
$msg' =$
$\langle A, enc(K_{AB}{}')(\langle A, hash_n(\langle w'\rangle, t)\rangle), e\rangle$

acknowledge

$\Delta(msg)$

$msg =$
$\langle B, enc(K_{AB})(\langle ack(\langle hash_n(\langle w\rangle, t)\rangle)\rangle)\rangle$
$msg' = \langle\rangle$

Pre-state variables are undecorated and hold the value of the variables before execution of the operation, whereas post-state variables are decorated with a prime '′' and denote the value of the variables after execution of the operation. The Object-Z symbol 'Δ' declares pre-state and post-state variables for each of the named variables, indicating that these variables may be changed by the operation.

Operation $requestKey$ corresponds to the sending of message 1 in the 'standard notation' description of the protocol shown in Section 3. This operation requires that there is no message currently in transit ($msg = \langle\rangle$). The post-state value of the message msg' consists of two items: Alice's identity, and an encrypted item containing Bob's identity encrypted using the key K_{AS} Alice shares with Sam. (Bob's identity is enclosed in a sequence before encryption since the encrypt function operates on messages, not on individual items.)

Operation $receiveKey$ corresponds to Alice both receiving message 2 from Sam and forwarding the key to Bob (message 3). Use of post-state variables $K_{AB}{}'$ and w' in the description of the incoming message msg models the way Alice receives and remembers these values. Alice treats the second item in the incoming message atomically, so we model it here simply as an encrypted item e. For the purpose of password generation, the maximum number t of times a password can be used is declared as a global constant below.

$t : \mathbb{N}_1$

Operation $acknowledge$ correponds to Alice receiving message 4 in order to authenticate Bob as the responding agent. The operation can occur only if the message in transit has the value shown. Since this is the final protocol operation, the message in transit is set to the empty sequence.

An agent in the server role determines which keys to use for decryption based on the agent identifiers received in the message. To capture this behaviour, the server requires a function key associating agent identifiers with keys. The server only requires one operation $giveKey$ that corresponds to both the receipt of message 1 and subsequent distribution of the session key in step 2.

Sam

$A, B : AID;\ w : SED$
$key : AID \pfun KEY$
$K_{AB} : KEY;\ T : TSP$
$msg : MSG;\ used : \mathbb{P}\, ITEM$

INIT

$msg = \langle\rangle \land used = \varnothing$

giveKey

$\Delta(msg, A, B, K_{AB}, w, T)$

$msg = \langle A', enc(key(A'))(\langle B'\rangle)\rangle$
$msg' = \langle enc(key(A'))(\langle B', K_{AB}{}', w'\rangle),$
$enc(key(B'))$
$(\langle A', B', K_{AB}{}', hash_n(\langle w'\rangle, t), T'\rangle)\rangle$
$\{K_{AB}{}', w', T'\} \cap used = \varnothing$
$used' = used \cup \{K_{AB}{}', w', T'\}$

Sam is required to keep items K_{AB}, w, and T, 'fresh' for each protocol instance. In Object-Z we specify this by keeping a record $used$ of previously used items. In operation $giveKey$ Sam chooses values (the post-state values) not already in this set to send in message 2. This is specified by ensuring that the set of new values intersected with the set of used values is null. Then he updates his record of used items to include these values using the union operator.

Bob's role is constructed below in a similar way, with one operation $receiveKey$ corresponding to Bob both receiving message 3 and sending an acknowledgement in message 4. Although we can use any words to identify

1530-0803/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

the various items, we choose to keep Object-Z identifiers consistent with the standard notation description for ease of reading. For example, '$H^t(w)$' is merely a variable name that identifies a hashed item. Similarly, this applies to 'K_{BS}' and 'K_{AB}'.

Bob

$A, B : AID;\ H^t(w) : HSH$
$K_{BS}, K_{AB} : KEY;\ T : TSP$
$msg : MSG;\ used : \mathbb{P}\ TSP$

INIT

$msg = \langle\,\rangle \land used = \varnothing$

receiveKey

$\Delta(msg, A, H^t(w), T, K_{AB})$

$T' \notin used \land used' = used \cup \{T'\}$
$msg = \langle A', enc(K_{AB}{}')(\langle A', H^t(w)'\rangle),$
$enc(K_{BS})(\langle A', B, K_{AB}{}', H^t(w)', T'\rangle\rangle$
$msg' = \langle B, enc(K_{AB}{}')(\langle ack(\langle H^t(w)'\rangle)\rangle)\rangle$

Normally Bob would check the timestamp against the current time to ensure that the message had been received in the correct time frame. However, we simplify this in our model by having Bob check that the received timestamp T' is not one he has seen before ($T' \notin used$). As part of the operation Bob also updates the set of used timestamps to include the latest one received.

The ideal model of an intruder is one based on the Dolev-Yao model [10] in which the intruder has complete control over messages sent between agents. An intruder with this capability can at any time *store* items from messages in transit or *send* a message composed of stored items. Assuming the intruder, Charles, does not know any keys for decryption of messages in transit, this behaviour is captured by the following Object-Z class.

Charles

$msg : MSG;\ stored : \mathbb{P}\ ITEM$

INIT

$msg = \langle\,\rangle \land stored = \varnothing$

store

$\Delta(stored)$

$stored' = stored \cup \text{ran}\ msg$

send

$\Delta(msg)$

$\text{ran}\ msg' \subseteq stored$

In Object-Z 'sequences' are actually functions from indices to sequence elements, so the operation *store* accesses the message items by taking the range of the message and adds them to the set of $stored'$ items. The *send* operation specifies that the items in the message msg' in transit will be a subset '$\subseteq$' of the previously *stored* items.

We now specify the *Protocol* in which specific instances of the four agent roles ($alice : Alice$, $bob : Bob$, $sam : Sam$ and $charles : Charles$) can communicate with one another. Agents' message channels msg are synchronised by way of a state invariant that ensures they are always equal. To complete the specification, each agent operation is restated as an operation of this class.

Protocol

$alice : Alice;\ bob : Bob$
$sam : Sam;\ charles : Charles$

$alice.msg = bob.msg$
$bob.msg = sam.msg$
$sam.msg = charles.msg$

$aliceRequestKey \mathrel{\widehat=} alice.requestKey$
$aliceReceiveKey \mathrel{\widehat=} alice.receiveKey$
$aliceAcknowledge \mathrel{\widehat=} alice.acknowledge$
$samGiveKey \mathrel{\widehat=} sam.giveKey$
$bobReceiveKey \mathrel{\widehat=} bob.receiveKey$
$charlesStore \mathrel{\widehat=} charles.store$
$charlesSend \mathrel{\widehat=} charles.send$

Thus we have specified precisely the behaviour of all the operations associated with the protocol. (The particular sequence in which the operations are performed is defined during the analysis.)

5 Modelling the protocol in SAL

Using the Object-Z schema calculus [11], specific scenarios can be verified to prove certain properties [15]. In recent research [25] the ability to apply model checking to Z-based specifications has been developed. In this paper we use our Object-Z specification as a basis for input to a model checker for automated analysis of the Kerberos-One-Time protocol.

SRI International's Symbolic Analysis Laboratory is a toolkit for analysis of state transition models. In particular, SAL's model checker [9] provides an automated means of verification, involving an exhaustive search of an abstract model to check that specified requirements always hold. The main disadvantage to model checking is that an exhaustive search can lead to the state explosion problem. Therefore, the model must be abstracted without removing too much essential information. The following sections illus-

IEEE COMPUTER SOCIETY

trate our translation of the Object-Z specification above into SAL's notation.

5.1 Simplifying the data types

In order to reduce the size of the state space to be explored, we restrict the data types to small non-recursive structures. For example, instead of using an infinite set of 'items' as in the Object-Z specification, in SAL we define the set and restrict it to a set of thirteen natural numbers. Additionally, we reserve `0` to represent the null item `XITEM`.

```
ITEM : TYPE = [0..12];
XITEM: NATURAL = 0;
```

Then each type of item is an even smaller subset of items as follows.

```
AID: TYPE = [1..3];
KEY: TYPE = [4..6];
SED: TYPE = [7..9];
TSP: TYPE = [10..12];
```

Instead of using a recursive structure to represent messages, we define two levels of messages since this is all that is required to model the particular protocol of interest. All items encrypted together or appearing in plaintext together we say belong to a sequence. For example, the ticket $\{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$ contains a sequence of five items. Since this is the longest sequence in the protocol, a sequence `SEQ` is defined as an array of five items. The null sequence `XSEQ` is specified by setting each position of the array to a null item.

```
SEQ : TYPE = ARRAY [1..5] OF ITEM;
XSEQ: SEQ  = [[i: [1..5]] XITEM];
```

Then each sequence belongs to a record also containing a key. The record identifies if a sequence of items is encrypted and, if so, with what key. If the key is `XITEM`, then the sequence is not encrypted. A null record `XREC` is a record containing null values for both the key and the sequence.

```
REC : TYPE =
 [# key: {x : ITEM |
     x = XITEM OR (x > 3 AND x < 7)},
    seq: SEQ #];
XREC: REC  =
 (# key := XITEM, seq := XSEQ #);
```

Finally, since message 3 is the longest message in the protocol, containing one plaintext sequence, and two encrypted sequences, we define a message `MSG` to be an array consisting of three records.

```
MSG: TYPE = ARRAY [1..3] OF REC;
```

5.2 Modelling protocol roles

To keep the SAL model consistent with the Object-Z specification, we again model each role as an individual class (or module in SAL). The state variables are declared in a similar way to those in the corresponding specification. The following module corresponds to the specification of Alice's role. Although we include some extra state information to control the order in which operations may occur to reduce analysis time, we have omitted this from the following modules for the sake of brevity.

```
alice: MODULE =
BEGIN
 GLOBAL msg: MSG
 GLOBAL alice_a: AID
 GLOBAL alice_b: AID
 GLOBAL alice_w: SED
 GLOBAL alice_kas: KEY
 GLOBAL alice_kab: KEY
 GLOBAL alice_keys: set{KEY;}!Set
INITIALIZATION [
 msg = XMSG AND
 set{KEY;}!empty?(used_keys)
 -->
 msg IN {x: MSG | true};
 alice_keys IN
  {x: set{KEY;}!Set | true};
]
TRANSITION [
 requestKey:
 msg = XMSG AND
 msg'[1] =
  (# key := XITEM,
     seq := [[i: [1..5]]
             IF i = 1 THEN alice_a
             ELSE XITEM ENDIF] #) AND
 msg'[2] =
  (# key := alice_kas,
     seq := [[i: [1..5]]
             IF i = 1 THEN alice_b
             ELSE XITEM ENDIF] #) AND
 msg'[3] = XREC
 -->
 msg' IN {x: MSG | true};
 [] receiveKey: ...
 [] acknowledge: ...
]
END;
```

In practice Alice need not keep a record of all received keys. However, we introduce a state variable `alice_keys`, purely for verification purposes, that is updated in the `receiveKey` operation to record all session

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

keys Alice has received. This allows us to state properties involving the history of keys received. Initially this set and the message in transit is empty as described by the INITIALIZATION predicate.

The message Alice sends in requestKey has two items: A and $\{B\}_{K_{AS}}$. (Once again, the primed variables are those representing the values after the operation has occured.) The first item msg'[1] is modelled by a record with no key XITEM and a sequence containing only Alice's identity alice_a. The second item msg'[2] is modelled by a record with Alice's key alice_kas and a sequence containing only Bob's identity alice_b. Since in our model each message must consist of three records, we state that the third part of the message is the null record XREC. The receiveKey and acknowledge operations are translated in a similar way.

The specification of Charles given in Section 4.3 allows him to store any number of items from transmitted messages and to construct any message from these stored items. However, we observe from the protocol that it is unlikely for a message containing random items to be accepted by any of the honest agents due to their operations' preconditions. Therefore, we simplify the intruder here by allowing him to store each of the entire four messages only, and to subsequently replay any of these messages at the appropriate time. This is modelled using an array MSGS of four messages.

```
MSGS: TYPE = ARRAY [1..4] OF MSG
```

Given this type, the intruder is described below. (The array of messages initially contains four empty messages.)

```
charles: MODULE =
BEGIN
 GLOBAL num: N
 GLOBAL msg: MSG
 LOCAL  stored: MSGS
TRANSITION [
   store:
    stored' = [[i:[1..4]]
               IF i = num THEN msg
               ELSE saved[i] ENDIF]
    -->
    stored' IN {x : MSGS | true};
[] send:
    msg' = stored[num]
    -->
    msg' IN {x: MSG | true};
[] nostore: ...
[] nosend : ...
]
END;
```

Counter num is used by the intruder to keep track of which message number the protocol is up to. This is used in the store operation as an index to store the current message at the appropriate position in the array. In the send operation this counter is used to replay the appropriate message for the current point in the protocol.

Once Charles has been introduced to the system, it is important to control the ordering of operations to prevent the intruder's operations from being executed indefinitely. We do this by allowing each one of the intruder's operations to be executed once at each point in the protocol at most. However, sometimes the intruder may wish to do nothing. For this reason, we also model two other operations, nostore and nosend, that simply skip to the next operation.

In order to simulate the protocol with the intruder present, we define the main module protocol to be the asynchronous composition of each of the modules.

```
protocol: MODULE =
 alice [] bob [] sam [] charles;
```

6 Verifying the protocol

SAL allows users to specify properties in linear temporal logic (LTL), and computation tree logic (CTL) [9]. LTL formulæstate properties about each linear path induced by the transition system modelled. Two LTL operators we use are:

- G(p), stating that predicate p is always true in every state; and
- X(p), stating that p is true in the next state.

When an invalid property is specified in LTL, a counterexample is produced revealing why the model is insufficient for its desired purpose.

6.1 Properties of the protocol

There are three properties we verify to ensure the protocol in Section 3 operates as desired. The first, sharedKey, ensures that at the end of each protocol instance, both Alice and Bob share the same key for K_{AB}.

```
sharedKey: THEOREM protocol |-
 G(state = end =>
   alice_kab = bob_kab);
```

This property states that it is always the case that when the protocol is in its end state, the value Alice has for the key she shares with Bob is the same as Bob's.

As mentioned in Section 3.3, Cimato states that the use of a session key should be restricted to a short time period,

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

avoiding the possibility of cryptanalytic attacks. Therefore, the second and third theorems ensure that once a new key has been received, it is not one that has been received before.

Earlier we introduced a set `alice_keys` of keys containing all session keys received by Alice. The following theorem `aliceFreshKey` compares the value of this set before and after Alice receives the new session key in message 2. Ideally, the values will always be different; otherwise we know the new key is not fresh.

```
aliceFreshKey: THEOREM protocol |-
 FORALL (keys : set{KEY;}!Set) :
  G((state = aliceReceivesKeyNext AND
     alice_keys = keys) =>
     X(alice_keys /= keys));
```

The third property is similar to the second only this time we ensure the value of the new session key in message 3 is fresh for Bob.

```
bobFreshKey: THEOREM protocol |-
 FORALL (keys : set{KEY;}!Set) :
  G((state = bobReceivesKeyNext AND
     bob_keys = keys) =>
     X(bob_keys /= keys));
```

6.2 Finding a counterexample

When the model checker runs, it proves that the first and third properties are valid; however, it presents a counterexample for the second property summarised by the following attack on the protocol in which Charles masquerades as both the Server and Bob.

1. $A \rightarrow S, C : A, \{B\}_{K_{AS}}$
2. $S \rightarrow A, C : \{B, K_{AB}, w\}_{K_{AS}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
3. $A \rightarrow B, C : A, \{A, H^t(w)\}_{K_{AB}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
4. $B \rightarrow A, C : B, \{H^t(w) + 1\}_{K_{AB}}$

$\vdots$

5. $A \rightarrow C(S) : A, \{B\}_{K_{AS}}$
6. $C(S) \rightarrow A : \{B, K_{AB}, w\}_{K_{AS}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
7. $A \rightarrow C(B) : A, \{A, H^t(w)\}_{K_{AB}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
8. $C(B) \rightarrow A : B, \{H^t(w) + 1\}_{K_{AB}}$

In the first four steps, Alice sets up a session key and password chain for secure communication with Bob. Charles observes and records the messages. When Alice is ready to establish a new session key for secure communication with Bob, she sends the message in step 5 which is identical to the message in step 1. During step 5, Charles intercepts the message from Alice. He then replays the message from step 2 masquerading as the Server (step 6). Alice has no way of checking whether this message is fresh. She responds in step 7. Charles intercepts this message and replays the message from step 4, successfully tricking Alice into believing that she has a secure session key and seed for secure communication with Bob (step 8). In actual fact, Alice has received a previously used session key. In the period between the two instances of the protocol above, Charles may have time to derive the stale session key, in which case the ramifications of the replay attack would be catastrophic.

In the situation above, Charles has successfully forced an old key to be used again. Even though the key is used for a short period of time once issued, Charles may have time to derive the key between the moment when the key was first issued and when he executes the replay attack. Alternatively, Charles could execute a denial-of-service attack on Alice, forcing her to respond repeatedly to bogus, but seemingly genuine, messages.

7 Fixing the protocol

Kerberos (Version 5) [13] includes a *nonce* in the first two messages of the protocol to prevent replay attacks. A nonce is a generated datum of low frequency that is difficult for others to guess. Given this property, an agent can identify the context of a message if it contains a nonce generated by them. For example, if Alice sends the randomly generated nonce N_A to the Server in the initial request, it is difficult for an intruder to reply with an appropriate response ($\{N_A, \ldots\}_{K_{AS}}$) without knowledge of the encryption key K_{AS}.

We observe that the Kerberos-One-Time protocol can also make use of such a nonce in order to prevent the replay attack we have discovered. The following is a representation of the protocol with this added security.

1. $A \rightarrow S : A, \{B\}_{K_{AS}}, N_A$
2. $S \rightarrow A : \{B, K_{AB}, w, N_A\}_{K_{AS}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
3. $A \rightarrow B : A, \{A, H^t(w)\}_{K_{AB}}, \{A, B, K_{AB}, H^t(w), T\}_{K_{BS}}$
4. $B \rightarrow A : B, \{H^t(w) + 1\}_{K_{AB}}$

Now Alice should be able to check that message 2 has not been replayed. If the message in step 2 contains a nonce different from the nonce sent in step 1, Alice will know that the message is not part of the current instance of the protocol and she will assume that a replay attack is in progress; thus, she will not continue the protocol.

Including this fix in the model requires some minor changes to the original specifications of Alice and Sam. Therefore, we take advantage of Object-Z's inheritance feature and provide only those parts of class specifications that are new or those operations that must be changed. In the following class, $FixedAlice$, this is signified by including the name of the original class, $Alice$, above the state schema.

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

The extra state variables are: N_A, declared as a nonce from an additional set NON of nonce items, and a set $used$ to record the nonces previously used by Alice. Initially, this set is empty.

FixedAlice

Alice

$N_A : NON$
$used : \mathbb{P}\, NON$

INIT

$used = \varnothing$

requestKey

$\Delta(msg, used, N_A)$

$msg = \langle\rangle$
$msg' = \langle A, enc(K_{AS})(\langle B\rangle), N_A'\rangle$
$N_A' \notin used \land used' = used \cup \{N_A'\}$

receiveKey

$\Delta(msg, K_{AB}, w)$

$\exists\, e : ENC \bullet$
$msg = \langle enc(K_{AS})(\langle B, K_{AB}{}', w', N_A\rangle), e\rangle \land$
$msg' =$
$\langle A, enc(K_{AB}{}')(\langle A, hash_n(\langle w'\rangle, t)\rangle), e\rangle$

In the operation $requestKey$, Alice now sends the nonce N_A' in the message msg'. This nonce is chosen at the time this operation is performed; Alice ensures it is one that has not been used before ($N_A' \notin used$) and she updates the set of used nonces to include this one. Alice checks in operation $receiveKey$ that the nonce she sent in message 1 is present in message 2. (Sam's model is updated similarly, simply to retrieve the new nonce from message 1 and to include it in message 2 for Alice.)

This time, after translating these changes into the SAL model, the model checker verifies that the three desired properties introduced in Section 6.1 hold. Hence, the fixed protocol is no longer susceptible to such replay attacks.

8 Conclusion

In this paper we have used the Object-Z formalism to present a formal specification of the Kerberos-One-Time protocol proposed for secure communication over the GSM mobile phone network. We then translated the specification into a model for analysis by the SAL model checker to confirm the existence of a suspected replay attack on the protocol. Given confirmation of the attack, we proposed a solution to the problem using an additional freshness identifier and used SAL to verify that this fix is effective. The resulting protocol is secure against such replay attacks and may be considered for use in GSM networks as originally suggested by Cimato.

Acknowledgements

We wish to thank Graeme Smith, Leonardo de Moura and John Rushby for their assistance using the SAL model checker, and the anonymous referees for their comments. This work was supported in part by Australian Research Council Linkage-Projects grant LP0347620, *Formally-Based Security Evaluation Procedures*.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, UK, 1996.

[3] A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Heám, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of the 17th International Conference on Computer-Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer-Verlag, 2005.

[4] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report TR 39, Digital Equipment Corporation, February 1989.

[6] S. Cimato. Design of an authentication protocol for GSM Javacards. In *Information Security and Cryptology — ICICS 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 355–368. Springer-Verlag, 2002.

[7] E. M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology*, 9(4):443–487, 2000.

[8] E. Cohen. Taps: A first-order verifier for cryptographic protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 144–158. IEEE Computer Society Press, 2000.

[9] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shanka, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer-Verlag, 2004.

[10] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

[11] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z.* Cornerstones of Computing. Macmillan Press Limited, UK, 2000.

[12] D. Gollmann. Analysing security protocols. In *Formal Aspects of Security*, volume 2629 of *Lecture Notes in Computer Science*, pages 71–80. Springer-Verlag, 2003.

[13] J. T. Kohl, B. C. Neuman, and T. Y. Ts'o. *The Evolution of the Kerberos Authentication Service*, pages 78–94. IEEE Computer Society Press, 1994.

[14] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[15] B. W. Long. Formal verification of a type flaw attack on a security protocol using Object-Z. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *4th International Conference of B and Z Users, ZB 2005*, volume 3455 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag, 2005.

[16] B. W. Long, C. J. Fidge, and A. Cerone. A Z based approach to verifying security protocols. In J. S. Dong and J. Woodcock, editors, *5th International Conference on Formal Engineering Methods, ICFEM 2003*, volume 2885 of *Lecture Notes in Computer Science*, pages 375–395. Springer-Verlag, 2003.

[17] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, February 1996.

[18] C. Meadows. Ordering from Satan's menu: a survey of requirements specification for formal analysis of cryptographic protocols. *Science of Computer Programming*, 50(1-3):3–22, 2004.

[19] J. K. Millen. The interrogator model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 251–260. IEEE Computer Society Press, 1995.

[20] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press, 1997.

[21] L. C. Paulson. Proving properties of security protocols by induction. In *Proceedings of 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 70–83. IEEE Computer Society Press, 1997.

[22] A. Renaud and P. Krishnan. An environment for specifying and verifying security properties. In *Proceedings of the Australian Software Engineering Conference*, pages 203–212. IEEE Computer Society Press, 2001.

[23] J. Rushby. The Needham-Schroeder protocol in SAL. Technical Report CSL Technical Note, SRI International, October 2003.

[24] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: The CSP Approach.* Addison-Wesley, 2000.

[25] G. Smith and L. Wildman. Model checking Z specifications using SAL. In H. Treharne, S. King, M. Henson, and S. Schneider, editors, *4th International Conference of B and Z Users, ZB 2005*, volume 3455 of *Lecture Notes in Computer Science*, pages 85–103. Springer-Verlag, 2005.

[26] D. Song, S. Berezin, and A. Perrig. Athena: a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1-2):47–74, 2001.

[27] J. M. Spivey. *The Z Notation : A Reference Manual*. Prentice Hall International Series In Computer Science. Prentice Hall, London, 1992.

[28] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*, pages 191–202, Dalas, Texas, February 1988.

[29] P. Syverson. A taxonomy of replay attacks. In *Proceedings of 7th IEEE Computer Security Foundations Workshop (CSFW'94)*, pages 187–191. IEEE Computer Society Press, 1994.

[30] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171. IEEE Computer Society Press, May 1998.

[31] The Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation*, 2.1 edition, Aug. 1999. ISO/IEC Standard 15408.

1530-0803/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY